High-performance Software Emulation of 1750A Processor

Kirk Reinholtz *

June 15, 1997

Abstract

We describe a software emulator of the MIL-STD-1750A architecture that executes 1750A code at a rate of about 4 MIPS on a Sun 200 MHz Ultra2 workstation, and effectively several times faster than that when application-specific optimizations are used. A number of optimization techniques were used, including binary translation and an unusual emulation of the 1750A timers and memory management unit. The performance technologies used within the emulator are for the most part applicable to the emulation of other processor architectures.

1 INTRODUCTION

We describe the implementation of a fast software emulator of the MIL-STD-1750A [1] processor architecture. The emulator executes 1750A binary opcodes just as a hardware processor would, and behaves just as a hardware processor would if it executed those opcodes.

The emulator is a performance-limiting component of the IISS series of spacecraft simulators[2]. A purpose of IISS is to emulate the "future" of operating spacecraft. Performance is thus critical, because it must be possible to stay ahead of the operating spacecraft, while also maintaining a margin for recovery in case of errors such that alternative futures must be explored or a simulation must be restarted.

The primary optimization technique we use is binary translation [3]: We translate (portions of) the 1750A code into the native instruction set of the host processor, so it can be executed much more efficiently. During the translation we also perform a number of specialized optimizations: Opcodes are decoded more extensively than is possible during runtime; Certain status bits are not computed if a subsequent instruction writes it again; and code to detect exceptions (e.g. arithmetic overflow) is not generated if it can be proven that either the exception can not occur or that it will be ignored.

Unlike other binary translators in the literature [3, 4, 5], we translate blocks of 1750A code into C++ functions, rather than directly into native code. The translated product is thus highly portable, and we don't have to duplicate the optimizations of a traditional compiler: the C++ compiler does those for us. Maintenance costs are reduced, and judging from an examination of the resulting code, performance is comparable to that of code generated by a custom compiler.

We do not require that all 1750A code be translated: the emulator will use translated blocks where available, resorting to interpretation as necessary. This fallback mechanism is an important feature of our emulator, since spacecraft code (and thus the code in our emulator) can be altered while it is executing.

We also optimized the MMU ("Memory Management Unit") algorithm, reducing it's amortized cost per access from dozens of lines of C++ to a couple of conditionals.

*The work described was performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

2 RELATED WORK

There is a long history of attempts to improve the performance of interpretive systems. One historical thread passes through the history of interpreted virtual machine Languages *de jour*: e.g. Smalltalk[4], APL[6], Forth[7], and most recently Java; and another through computer systems that support execution of programs built for other computer systems[8, 3, 9].

Most of the optimizations we use have been used previously, in significant undertakings of IBM [8] and later by Digital[3]. Unique to our system is the use of an existing compiler system to perform traditional optimizations, coupled with domain-specific optimizations performed by our translator. Our approach greatly reduces the cost of the development and maintenance of the emulator, yet provides the sophisticated optimizations of other approaches.

3 ARCHITECTURE

The emulator implements all MIL-STD-1750A features, and a number of vendor-specific[10] and hfll,-S'1'])-175011[11] extensions. Feature sets (e.g. breakpoints, instruction tracing, vendor-specific extensions) may be collected into personalities, all of which share the same CPU state and so can be switched during execution of the emulator. Thus features unnecessary to a particular application, or even to an execution phase of the same application, do not impact the performance of the application.

The 1750A emulator is designed to exhibit fidelity sufficient that it is not possible to write a 1750A program that can discriminate between a hardware processor anti the emulator. To this end, the emulator is tested with the Wright-Patterson AFB 1750A test suite known as 1750A VSW (Verification Software), version 2.4.1: The same tests that are used to certify hardware implementations of the 1750A architecture. The suite has several hundred test sets, and thousands of individual test vectors, that exercise most aspects of the 1750A architecture.

The emulator can be interfaced with external components. Each page of memory is either mapped directly into RAM, for ultimate performance, or more generally into an i/o dispatch unit that individually associates each word in the page with some address in some external model or some address in RAM. Such accesses call read/write functions in the external device, and so can envoke arbitrary computations, thus providing a facility that has recently been termed "Active Memory"[12].

The emulator architecture is based upon the usual interpreter loop, as outlined in approximate and simplified form in figure 1). Registers are implemented as an array R. Union structures are used to decode and extract various bit fields from the instruction.

Certain opcodes (selected from the set of unused opcodes) e.g. 0x5B in figure 1 cause functions to be called, rather than opcodes to be interpreted. This is the basis of the binary translation optimization.

```
int execute (int cycles _to_do)
  int c_done = O
  tcache = do_timers(c_done)
  while(c_done < cycles_to_do)
    check_interrupts()
    instr = MemRead(R[IC])
    R[IC] += length_of(instr)
    c_done += cycles_of(instr)
    switch(instr.opcode)
      ...
      case 0x5b:
        c_done += fast [instr.val] (this)
        break
      ...
    endswitch
    if(c_done >= tcache)
      tcache = do_timers(c_done)
    endif
  endwhile
  return c_done
```

Figure 1: Emulator algorithm

4 OPTIMIZATIONS

4.1 Binary Translation

Binary translation refers to the act of converting a sequence of binary code from one format (e.g. 1750A) to another (e.g. SPARC), such that the latter fragment of code behaves exactly as the former with respect to the visible state of the former. In other words, the SPARC code manipulates the emulated 1750A registers and memory exactly as does the 1750A code, but it executes directly on the SPARC processor rather than within an interpreter.

Binary translation is often used to increase the performance of interpretive systems, and does so on three basic fronts: It eliminates some interpretive overhead, since that overhead becomes subsumed by the SPARC processor itself; Some things may be precomputed during the translation, for example register fields within the 1750A opcode; and certain other optimizations become possible, since a sequence of opcodes is translated and thus certain flow-based information can be used that is *not* available to an interpretive architecture.

Our translator operates in two phases: First, it translates the 1750A code sequences into C++ functions with equivialent semantics, but with certain domain-specific optimizations applied. Then the host C++ compiler translates the C++ functions into native code, during which all of the usual compiler optimizations are performed. The output of the first stage is designed to make it easy for the compiler to optimize the code.

As a last resort, the C++ functions may be edited manually before compilation, though we have not found this necessary. We in fact tried to optimize some of the functions beyond what is done by the translator (e.g. manual dataflow analyis), but found the results of minimal consequence. From this we conjecture that there isn't much more performance to be gained from increasing the sophistication of the translator optimizations.

The translator performs a number of optimizations specific to the 1750A, outlined here.

Compile-time opcode decoding The interpreter must extract certain information from each opcode each time the opcode is encountered. For example, some bitfields in the opcode specify which index and other registers to use. Since the value of those fields does not change during program execution, the translator dots the decoding during the translation, providing two benefits: We avoid the cost of runtime decoding; and the compiler has more opportunity to optimize, since e.g. array indexes into the register data structure become known at compile-time.

Defer CC writes In a dynamic sense, many 1750A opcodes write to the condition codes, but many less read the results. We found it profitable to supress the code that writes the condition codes if there is a subsequent write before a subsequent read.

Don't compute exceptions Many 1750A arithmetic operations check for exceptional conditions, e.g. overflow and underflow. It is costly to perform the checks in software, so the translator supresses generation Of the code that perforn IS the checks if it can be known that either the exception will be ignored or can not occur.

Branches converted to C++ conditionals In retain cases we translate 1750A conditional branches directly into C++ code, which virtually eliminates the cost of emulating the branch.

4.2 Memory Management Unit

The 1750a MMU takes as input a 16-bit logical address, address type (instruction or operand), access mode (read or write), address state, and perhaps data to be written. It validates the operation against the current protection mode of the CPU, and if the operation is legal it performs the operation, otherwise it sets certain status bits to indicate what type of violation occured, which may subsequently cause an interrupt to be asserted.

The algorithm required to compute the physical address and legality of the operation is substantial, and would take many instruction-times to compute were it emulated for each memory access. However,

the CPU state items that) affect the outcome of the computation, other than the logical address, do not often change.

We therefore pre-compute the legality of access to each logical page. If the logical page refers to physical RAM and allows both reads and writes, we put the physical address directly into a MMU cache such that the physical RAM location can be quickly located, If the logical page is fully accessable, or refers to something other than RAM (e.g. a hardware i/o register) then the MMU cache slot is set to zero, which causes the! full MMU algorithm to be performed upon the address. Thus most operations (legal reads and writes) execute very quickly. Only unusual conditions (illegal accesses or hardware i/o) execute slowly.

The MMU cache is implemented as an array indexed by logical page number that contains either zero, which indicates that the full MMU algorithm must be applied, or au address such that the following expression yields the address in host memory to which the logical address refers.

```
paddr = cache [lpage]+laddr
```

To write a value to the address requires only tile following operation. This computation car ı be performed iu a few host instruction times, as opposed to many tens or perhaps hundreds of host instruction times required to compute the full MMU algorithm.

```
if cache[tmp=(laddr >> 12)] != O then
  *(cache[tmp]+laddr) = value
else
  full_MMU_write(laddr,value)
endif
```

Note that the value iu the cache array is not just the offset to the page, but includes compensation so that the logical address may simply be added to the cache value. Otherwise you would have to expend more cycles extracting the offset, from the logical address.

The contents of the cache array is initialized every time the address state changes, and every time the page registers for the relevant page are modified.

4.3 Lazy evaluation of arithmetic condition codes

Many arithmetic and other operations return not only a result, but set cert ain condition flags as well. The 1750A has four condition codes: carry/borrow, positive, zero, negative. There is a bit in the CS field of the SW register for each of these conditions.

Consider an opcode that computes R0ₐ - R0 + R1. The following code fragment shows how this would be implemented w/o lazy evaluation:

```
R0 = R0 + R1
if (RO > 65535)
     SW |= 0x8000
     RO -= 65536
end if
if(R0 >  O )
  SW|= 0X4000
elseif(R0 >  O )
  Sw |=0x1000
else
   (RO == 0)SW|= 0x2000
endif
```

Note that this requires u significant amount of code to compute the condition codes.

Instead, we use the same technique as outlined in [3]: Simply save the result of the operation (in this case R0) and only conduct the checks should a later operation require the results, Thus our code for the above looks like this:

```
RO = R0 + RI
if(R0> 65535)
     SW |= 0x8000
     RO -= 65536
endif
CSlazy = RO
```

Here is how a code fragment would check that the result of a previous operation was positive, for both fragments above. (A) is for the naive implementation, (B) is for our implementation:

```
A: if(SW&0x2000) . . .
B: if(CSlazy == O) . . .
```

You can see that our method does not impose significant burden on this computation, either, and so does provide a performance improvement. We use this same technique in the interpreter and the binary translator.

In the 1750A it is possible to force the condition codes to represent conflicting states, e.g. negative and positive, simultaniously. For this reason C Slazy carries a "valid" bit (not shown in the examples) that, if clear, indicates that the actual condition codes must be examined. We have not observerd this situation in practice, but we do see in in the VSW tests.

4.4 Floating-point

Floating point is used heavily by many aerospace applications, for example navigation and attitude control. Performance is thus important, and optimized as outlined in this section.

MIL-STD-1750A specifies a particular data format and implementation of floating point, unfortunately not ANSI/IEEEStd **754-1985.** (if it were, we could straightforwardly use the workstation IEEE floating-point hardware that is nearly the universal standard today.) Rather than implement the floating point as many lines of C++ code following the algorithm is specified in the standard, we instead translate the operands, during runtime, into IEEE 754 format (or, more generally, the native format of the host computer), perform the operation on the workstation hardware, then convert the results back to 1750A format.

We found that particular attention must be paid to the details of conversion and overflow, underflow and rounding. Fortunately the VSW tests include a large number of vectors to explore the behavior of the floating point implementation, as otherwise it is likely that we would have made undetected mistakes.

4.5 Timers

The **1750A** has two timers, called timer **A** and timer B. Each has a 16-bit counter, and a reload value. The A and B counters are incremented (i.e. "tick") every 10 and 100 microseconds, respectively. When a counter rolls over to zero. it causes an interrupt to be posted, and the counter is reloaded with its reload value.

Timer emulation must be fast, since it executes every instruction, and it must not drift, since it is possible for software to compare the timer rate against an external source.

We express the timer rates in rational form, "n" CPU cycles per "m" counter ticks so that most clock rates can be exactly expressed and therefor clock drift relative to timer ticks can be prevented. Vis timer A, 1.25 MHz is 25 instructions per 2 ticks. Floating point does not eliminate the problem, and is much slower than integer arithmetic on most workstations.

Figure 2 outlines a simple but correct implementation of a 1750A timer. "ace" and "counter" are 16-bit unsigned integers. "olt," is the number of instruction cycles by which the timer is to be advanced. The algorithm costs at, least a couple of conditional branches and a couple of arithmetic operations each time this code fragment is executed, and it must be executed pretty much after each in struction is executed. It is apparent that this is a significant performance bottleneck.

```
if(timer_enabled)
  acc += m*dt
  dt = O
  while(acc >= n)
    acc -= n
    counter += 1
    if(counter == O)
      interrupt ()
      counter = initial_value
    endif
  endwhile
fi
```

Figure 2: Naive timer algorithm

To improve the performance of this algorithm, we simply advance (it to the first pre-computed point in the future at which either the counter will roll to zero, or just before and arithmetic overflow would occur. Also, if the CPU observes or changes the state of the timer registers, then dt must be written to

those registers before the observation or change takes place. The timer overhead thus amortized to about one compare per interpreter cycle.

Figure 3 shows fast algorithm. Note that the body of the algorithm executes only when dt has advanced far enough to cause a timer interrupt to occur (as opposed to every few instructions in the naive implementation), so the loop overhead imposed by the timer is dominated by the single comparson of dt to tcache.

```
if(dt >= tcache)
  old_cnt = ((int)counter)
  counter = counter + (((int)acc)+dt*m)/n
  acc = (((int)acc)+dt*m)%n
  if(counter < old_cnt)
    interrupt
    counter += initial_value
  endif
  dt = 0
endif
```

Figure 3: Fast timer algorithm

Finally, figure 4 lists the algorithm that computes the value of tcache at which the timer will next cause an interrupt.

```
d2 = (n*(65536-counter)-acc-dt*m+m-1) /m
d3 = (65535-acc)/m-dt
tcache = min(d2,d3)
```

Figure 4: Computation of tcache

5 FURTHER WORK

The current system uses a static database that associates each block with an address in each version of software. We conjecture that fielded flight software dots not change often, and when it dots change, it doesn't change a lot. This suggests that most blocks are probably still valid, though their installation addresses may have changed. We propose to dynamically place blocks, perhaps based upon a signature of the opcode sequence to which each applies, so that the system gracefully adapts to new versions of the flight software. It would still be necessary to add to the block library as time goes on, but for normal small changes it would not be necessary to pay attention to the binary translation system.

We would like to revisit the MOV instruction. If, for example, we could prove that a certain instance of MOV has both its source and destination in RAM, we could then replace the instance with a native memcpy(). This would improve performance two ways: (1) MOV becomes faster; and (2) that particular MOV would not terminate a block translation.

6 CONCLUSION

We have constructed an "industrial strength" 1750A emulator that operates at several MIPS on real code (not a simple benchmark) on circa '97 workstations, and can deliver effective performance well beyond that number using simple application-specific optimizations.

A OVERVIEW OF 1750A

MIL-STD-1750A specifies an instruction set architecture ("ISA"). It explicitly does not address physical concerns, e.g. speed, weight, power, i/o capabilities, and so on. The intent is that code is portable amongst compliant processor implimentations.

The 1750A specifies a sixteen-bit data word, sixteen-bit logical addresses, and twenty (D1' more) bit physical addresses. '1'here are sixteen address st ates, each of which maps a set of 4KWord pages into the physical address space. Each page is marked either read-only, execute-only, or read/writ,e. Further, the physical address space has write protection in 1KWord units. The 1750A has floating point instructions, using 175( )A-specific floating-point data formats and algorithms.

The once-proposed MI 1,-S'1'1)-1750}1, and most vendor extensions to MIL-STI)-1750A, basically add some additional instructions, and accomodation o f

larger physical address space, to the base 1750A architecture.

References

[1] Military Standard Sixteen-Ilit Instruction Set Architecture. Department of Defense, July 1980. MIL-STD-1750A (USAF).

[2] A. Morrissett et al. Multimission I ligh Speed Spacecraft Simulation For The Galileo and Cassini Missions. In *A IA A Computing in A erospace Conference 9th, San Diego, CA, October* 19-21, *1993*. American Institute of Aeronautics and Astronautics, October 1993.

[3] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S. G. Robi nson. Binary Translation. *Communications of the ACM*, 36(2):69 81, 1993.

[4] L.P. Deutsch and A.M. Schiffman. Efficient Implementation of the SmallTalk-80 System. *A CM Symposium on Principles of Programming Languages, January*, TBD(TBD):297 302, 1984.

[5] A.B. Bergh, K. Keliman, D.J. Magenheimer, and J.A. Miller. HP 3000 Emulation 011 1 [1' Precision Architecture Computers. *Hewlett-Packard Journal*, TBD(TBD):87- 89, 1997.

[6] H.J. Saal and Z. Weiss. A Software 1 ligh Performance APL Interpreter. *APL Quote Quad*, 9(4):74 81, June 1979.

[7] P.M. Kogge. An Architectural Trail to Threaded-Code Systems. *Computer*, 15(3):22 32, March 1992.

[8] C.May. MIMIC: **A** Fast System/370 Simulator. *SIGPLAN*, 22(7):1- 13, 1987.

[9] L. Wirbel. D OS-to-Unix Compiler. *Electronic Engineering Times*, pages 8 3 end, March 1 4 1988.

[10] Generic VHSIC Spaceborne Computer (Phase 11), 1989.

[11] Proposed SAE Standard sixteen-bit computer set, architecture. Department of Defense, June 1989. DRAFT MIL-STD-1750B (USAF).

[12] A. I{. Lebeck and D.A. Wood. Active Memory: A new Abstraction for Memory Systems Silllul:l-tion. *A CM Transactions on Modeling and Computer Simulation*, 7(1) :42 76, January 1997.